Parallel Computing for Science & Engineering

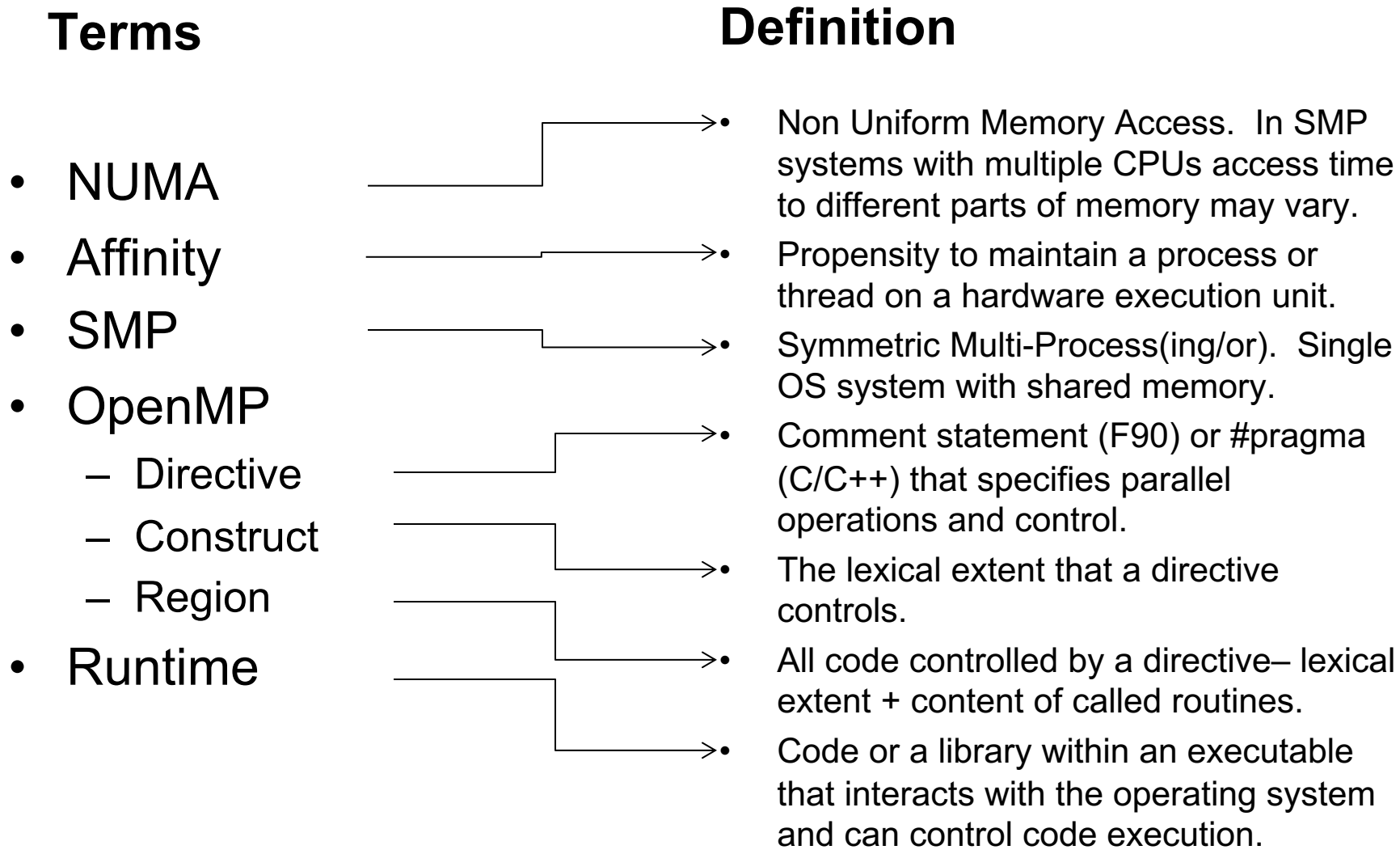
02/23/23 to 3/9/23

Instructors

Lars Koesterke, TACC

Victor Eijkhout, TACC

Scientific Computing Terminology



OpenMP-- Overview

- Standard is ~25 years old. Mature language
- The “language” is easily comprehended.
  You can start simple and expand.
- Light Weight from System Perspective
- Very portable –GNU and vendor compilers.
- Spend time finding parallelism can be the most difficult part. The parallelism may be hidden.
- Writing Parallel OpenMP code examples is relatively easy.
- Developing parallel algorithms and/or parallelizing serial code is much harder.
- Expert level requires awareness of scoping and synchronization.
- Expansion into other performance relevant areas like: thread pinning and memory pinning

OpenMP --- Shared Memory

- Shared Memory systems:
  - One Operating System
  - Instantiation of ONE process
  - Threads are forked (created) from within your program.
  - Multiple threads on multiple cores

What is OpenMP (Open Multi-Processing)

- **De facto standard for Scientific Parallel Programming on Symmetric Multi-Processor (SMP) Systems.**

- **It is an API (Application Program Interface) for designing and executing parallel Fortran, C and C++ programs**
  - Based on threads, but
  - Higher-level than POSIX threads (Pthreads) (http://www.llnl.gov/computing/tutorials/pthreads/#Abstract)
- **Implemented by:**
  - **Pragmas/comments in code**
  - **Runtime Library (interface to OS and Program Environment)**
  - **Environment Variables**
- **Compiler option required to interpret/activate directives.**

- **http://www.openmp.org/ has tutorials and description.**
- **Directed by OpenMP ARB (Architecture Review Board)**

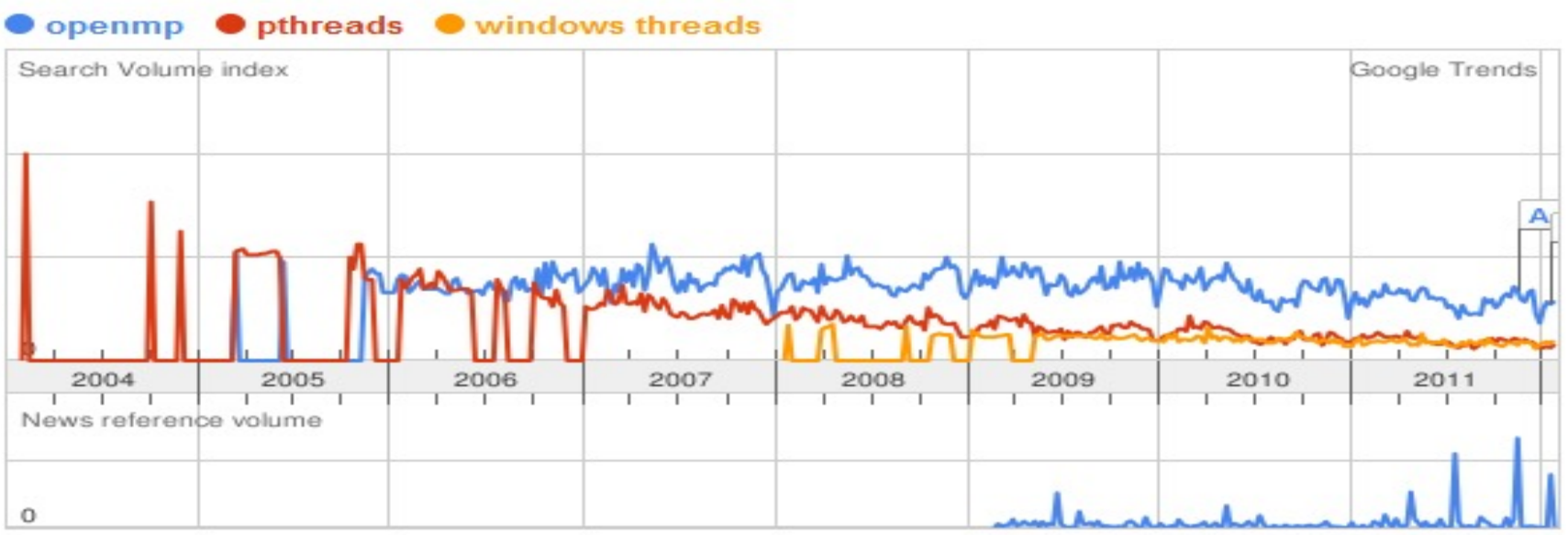
OpenMP History

- Primary OpenMP participants

  AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, PGI, Oracle, MS, TI, CAPS, NVIDIA
  ANL, LLNL, cOMPunity, EPCC, LANL, NASA, ORNL, RWTH, TACC

- OpenMP Fortran API, Version 1.0, 1997
- OpenMP C API, Version 1.0, 1998
- OpenMP 2.0 API for Fortran, 2000
- OpenMP 2.0 API for C/C++, 2002
- OpenMP 2.5 API for C/C++ & F90 2005
- OpenMP 3.0 Tasks May 2008
- OpenMP 3.1 July 2011
- OpenMP 4.0 **Affinity, Devices, Depend, SIMD** July 2013
- OpenMP 4.5
- OpenMP 5.0

OpenMP History



OpenMP 3.0: The World is still flat, no support for NUMA (yet)!
OpenMP is hardware agnostic, it has no notion of data locality.
The Affinity problem: How to maintain or improve the nearness of threads and their most frequently used data.
Or:
Where to run threads?
Where to place data?

http://terboven.wordpress.com/

Thread binding was added in OpenMP 4.0

Advantages/Disadvantages of OpenMP

- Pros
  - Shared Memory Parallelism is easier to learn.
  - Coarse-grained or fine-grained parallelism
  - Parallelization can be incremental
  - Widely available, portable
  - Converting serial code to OpenMP parallel can be easier than converting to MPI parallel.
  - Shared-Memory hardware is prevalent now.
    - Supercomputers **and** your desktop/laptop (and your phones)
    - GPUs (Graphics Cards), Multi-core CPUs
  - Complements MPI and enables full core utilization
- Cons
  - Scalability limited by memory architecture.
  - Available on Shared-Memory systems “only”.
  - Beware: “Upgrading” large serial code may be hard.

OpenMP Parallel Directives

Supports parallelism by Directives in Fortran, C/C++,…

Unlike others that require base language changes and constructs

Unlike MPI which supports parallelism through communication lib.

OpenMP implementation through the compiler

Compiler optimizes OpenMP code

Processes on a shared-memory System

- The OS starts a process
  - One instance of your computer program, the “a.out”
- Many processes may be executed on a single core through “time sharing” (time slicing).
  - The OS allows each process to run for awhile.
- The OS may run multiple processes concurrently on different cores.
- Security considerations
  - Independent processes have no direct communication (exchange of data) and are not able to read another process’s memory.
- Speed considerations
  - Time sharing among processes has a large overhead.

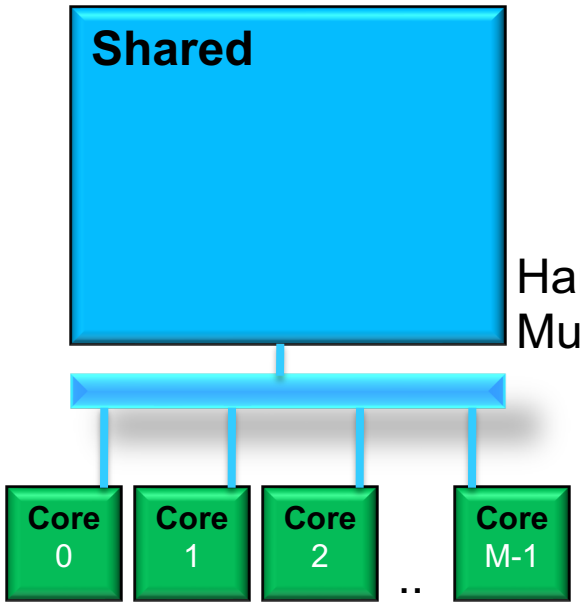
OpenMP Threads

- Threads are instantiated (forked) in a program
- Threads run concurrently*
- All threads (forked from the same process) can read the memory allocated to the process.
- Each thread is given some private memory only seen by the thread.

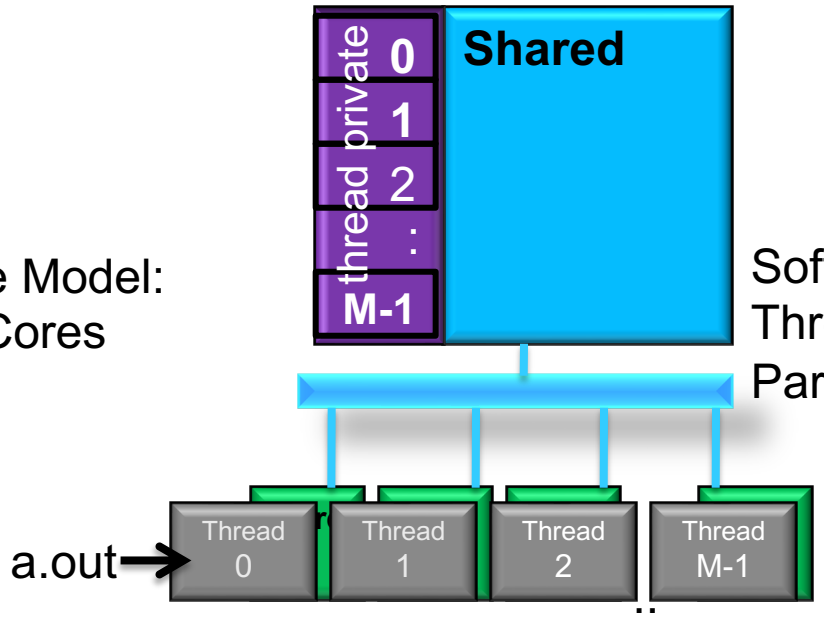
- *When the # of threads forked exceeds the # of cores, time sharing (TS) will occur. Usually you would not do this. (But TS with user threads is less expensive than TS with processes).

- Implementation of threads differs from one OS to another.

Programming with OpenMP on Shared Memory Systems



Hardware Model: Multiple Cores



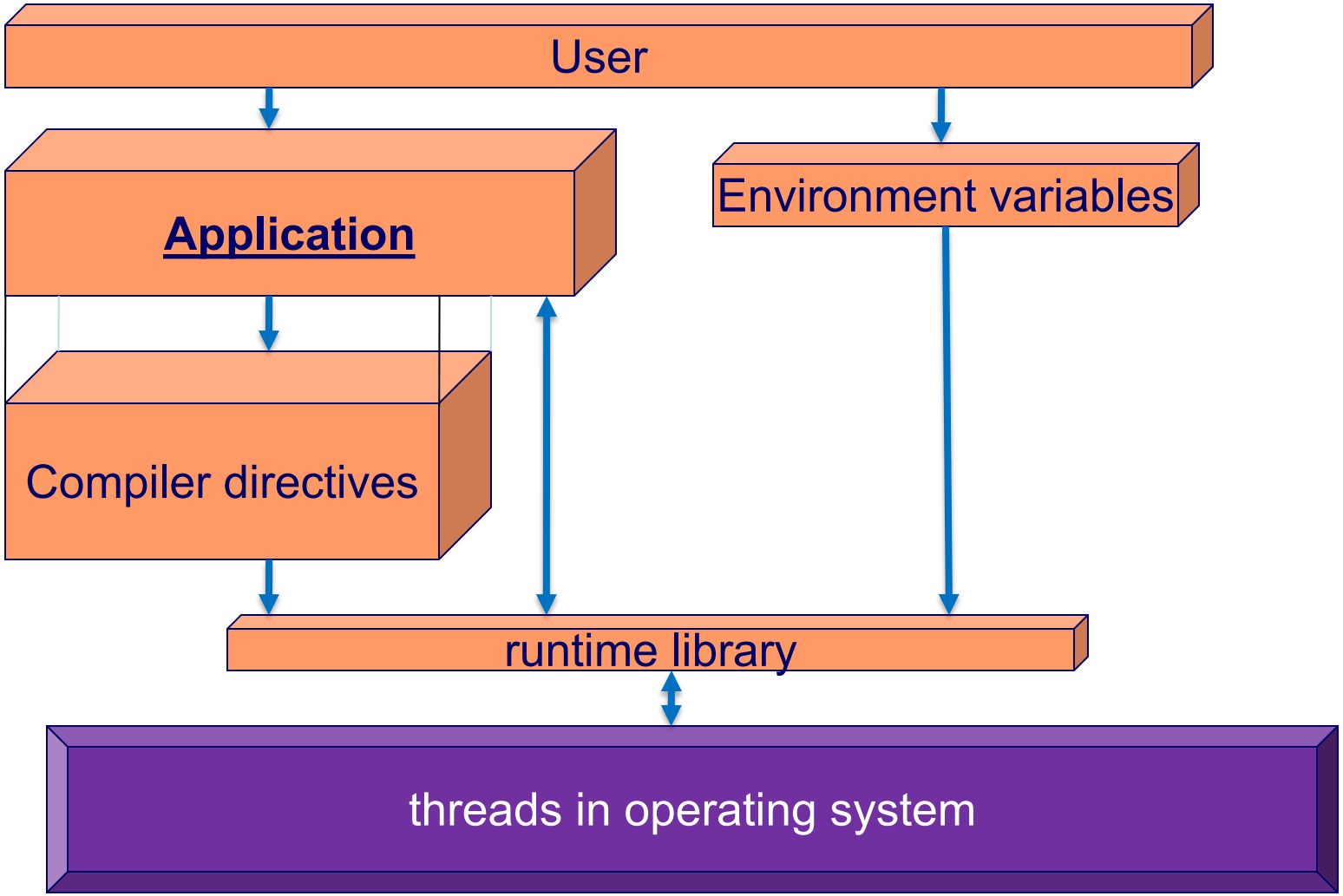
Software Model: Threads in Parallel Region

M threads are usually mapped to M cores.

Shared = accessible by all threads

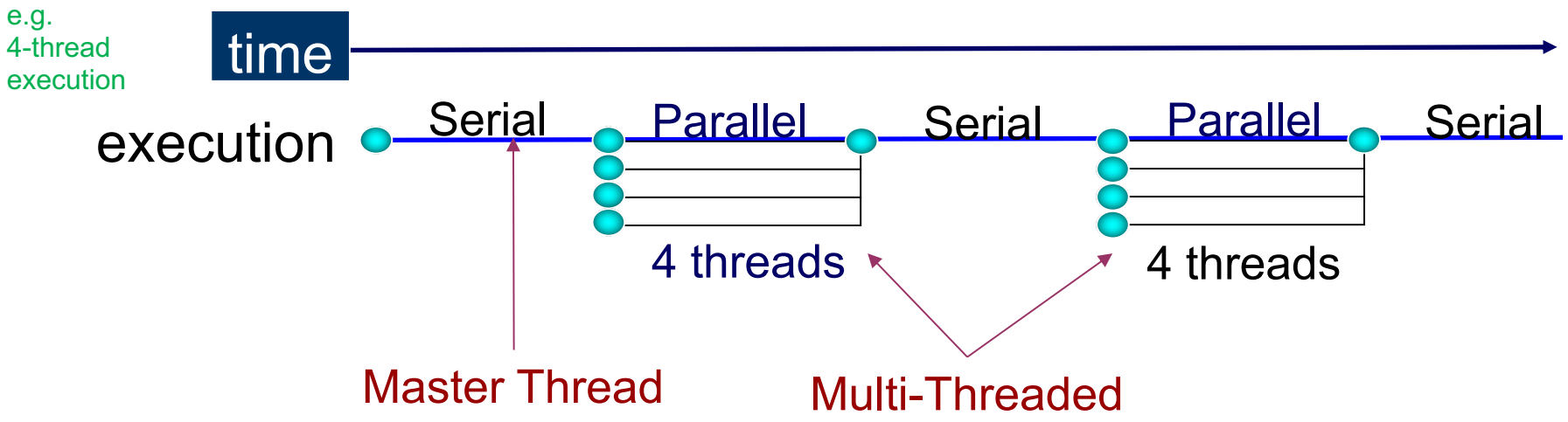
x = private memory for thread x

OpenMP Architecture



OpenMP Fork-Join Parallelism

- **Programs begin as a single process**: master thread
- Master thread executes in serial mode until the parallel region construct is encountered
- Master thread **creates (forks)** a team of parallel **threads** that simultaneously execute tasks **in a parallel region**
- After executing the statements in the parallel region, team threads synchronize and terminate (**join**) but master continues



Intermission

```
a = b + c
```

```
a(i) = b(i) + c(i)
```

```
loop (i) from 1 to n
  a(i) = b(i) + c(i)
end loop
```

Executing in Parallel

How does this work?

```
loop (i) from 1 to n
  a(i) = b(i) + c(i)
end loop
```

You want to execute this with the help of your buddies

What do you have to do?

Executing in Parallel

How does this work?

```
loop (i) from 1 to n
  a(i) = b(i) + c(i)
end loop
```

You want to execute this with the help of your buddies

What do you have to do?

1. Assemble a team
2. Divide up the work
3. Everybody works on their share of the loop
4. Wait for everybody to finish
5. Disassemble team

Executing in Parallel

How does this work?

```
loop (i) from 1 to n
  a(i) = b(i) + c(i)
end loop
```

You want to execute this with the help of your buddies

What do you have to do?

	'Plain english'	'OpenMP lingo'
1.	Assemble a team	Fork threads
2.	Divide up the work	Work sharing
3.	Everybody works	*Work* in parallel
4.	Wait for everybody	Barrier
5.	Disassemble team	Join threads

OpenMP Syntax

- OpenMP Directives: **Sentinel**, **construct** and **clauses**

  #pragma omp ***construct*** **…** **C**

  !$omp ***construct*** **…** **F90**

- **Example for a loop**

  #pragma omp parallel num_threads(4) **C**

  !$omp parallel num_threads(4) **F90**

Loop Example

```
    ...
1.  !$omp parallel
1.
2.  !$omp do
3.  do i=1,n
4.    a(i) = b(i)+c(i)
5.  end do
6.  !$omp end parallel
```

```
...
#pragma omp parallel
{
#pragma omp for
  for(i=0;i<n;i++){
    a[i] = b[i]+c[i];
}
}
```

Let's identify the 5 steps:
Fork threads, work-sharing, 'actual work', barrier, join threads

Loop Example

```
    ...
1.  !$omp parallel
1.
2.  !$omp do
3.  do i=1,n
4.    a(i) = b(i)+c(i)
5.  end do
6.  !$omp end parallel
```

```
...
#pragma omp parallel
{
#pragma omp for
  for(i=0;i<n;i++){
    a[i] = b[i]+c[i];
}
}
```

Let's identify the 5 steps:
Fork threads, work-sharing, 'actual work', barrier, join threads

Forking/joining threads

Work-sharing with implied barrier at the end

2nd Loop Example

```
    ...
!$omp parallel

!$omp do
do i=1,n
  a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n/2
  a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
  a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n/2;i++){
  a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

How many barriers are there?
How many barriers do we need?

2nd Loop Example

```
    ...
!$omp parallel

!$omp do
do i=1,n
  a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n/2
  a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
  a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n/2;i++){
  a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

How many barriers are there?
How many barriers do we need

1 parallel region

2 work-sharing constructs

2nd Loop Example

```
    ...
!$omp parallel

!$omp do
do i=1,n
  a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n/2
  a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
  a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n/2;i++){
  a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

How many barriers are there?
How many barriers do we need

There is a barrier at the end of the parallel region

There is an implied barrier at the end of the every 'work-sharing' construct'

2nd Loop Example

```
    ...
!$omp parallel

!$omp do
do i=1,n
  a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n/2
  a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
  a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n/2;i++){
  a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

How many barriers are there?
How many barriers do we need

There is a barrier at the end of the parallel region

There is an implied barrier at the end of the every 'work-sharing' construct'

Total number of barriers: 3

TACC

2nd Loop Example (modified)

```
    ...
!$omp parallel

!$omp do
do i=1,n
  a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n
  a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
  a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n;i++){
  a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

How many barriers are there?
How many are necessary to ensure correct results?
How many barriers do we need?

2nd Loop Example (modified)

```
    ...
!$omp parallel

!$omp do
do i=1,n
  a(i) = b(i)+c(i)
end do

!$omp do
do i=1,n
  a(i) = a(i)*d(i)
end do
!$omp end parallel
```

```
...
#pragma omp parallel
{
#pragma omp for
for(i=0;i<n;i++){
  a[i] = b[i]+c[i];
}

#pragma omp for
for(i=0;i<n;i++){
  a[i] = a[i]*d[i];
}
}
```

How many parallel regions?
How many work-sharing constructs?

How many barriers are there?
How many are necessary to ensure correct results?
How many barriers do we need?

2 barriers are needed

First: to ensure correct results
Second: to join the threads

End of Intermission

Examples of Parallel Computing

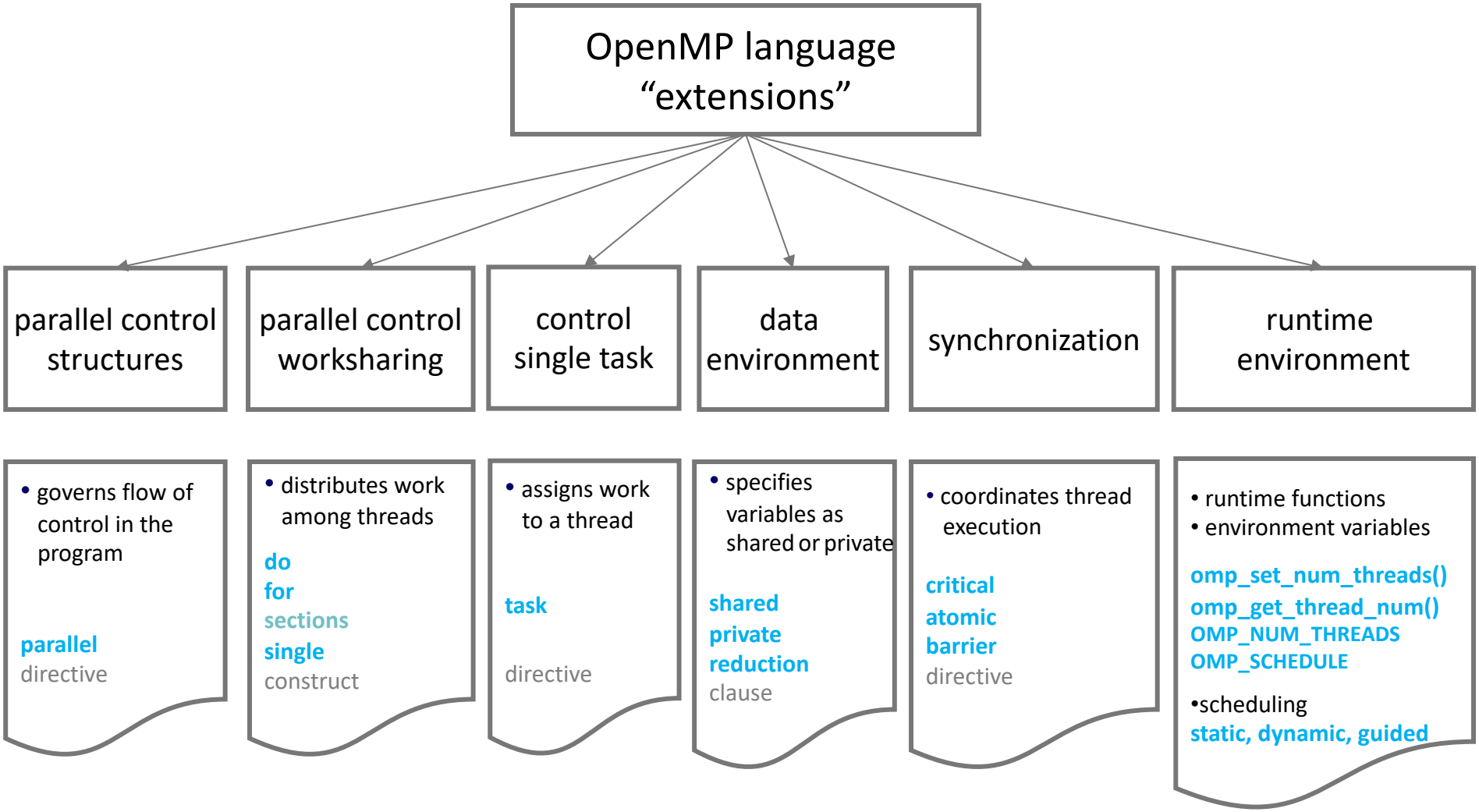
- Concurrent execution of computational work (tasks).
  - Tasks execute independently
  - Variable Updates must be mutually exclusive
  - Synchronization through barriers

```
1   // We use loops for
    // repetitive tasks
2   for (i=0; i<N; i++){
3
4     a[i] = b[i] + c[i];
5   }
```

```
1   // We often update
    // variable(s)
2   for (i=0; i<N; i++){
3
4     sum = sum + b[i]*c[i];
5   }
```

Parallel directives go here …

OpenMP Constructs



Directive Sentinels: "!$omp" and "#pragma omp" not shown.

OpenMP Syntax

- OpenMP Directives: **Sentinel**, **construct** and **clauses**

  #pragma omp ***construct*** **[*clause* [[,]*clause*]…]** **C**

  !$omp ***construct*** **[*clause* [[,]*clause*]…]** **F90**

- Example

  #pragma omp **parallel num_threads(4)** **C**

  !$omp **parallel num_threads(4)** **F90**

- Function prototypes and types are in the file:

  #include <omp.h> **C**

  use omp_lib **F90**

- Most OpenMP constructs apply to a “structured block”, that is, a block of one or more statements with one point of entry at the top and one point of exit at the bottom

OpenMP Directives

- **OpenMP directives begin with special comments/pragmas that open-aware compilers interpret. Directive sentinels are:**

F90	**!$OMP**
C/C++	**# pragma omp**

Syntax: ***sentinel*** **construct** *clauses* *defaults used when no clauses present*

Fortran

```
!$OMP parallel
  ...
!$OMP end parallel
```

C/C++

```
# pragma omp parallel
  {...}
```

Fortran Parallel regions are enclosed by **enclosing directives.**

C/C++ Parallel regions are enclosed by **curly brackets.**

Parallel Region

```
    ...
1  !$omp parallel
2        code statements
3        call work(…)
4   !$omp end parallel
```

```
...
#pragma omp parallel
{  code statements
   work(…)
}
```

Line 1 Team of threads formed.

Lines 2-3 This is the parallel region
Each thread executes code block and subroutine call or function.
No branching (in or out) in a parallel region.

Line 4 All threads synchronize at end of parallel region (implied barrier).

Replicated work or work-sharing?

Parallel Region

```
    ...
1  !$omp parallel
2        code statements
3        call work(…)
4   !$omp end parallel
```

```
...
#pragma omp parallel
{  code statements
   work(…)
}
```

Line 1 — Team of threads formed.

Lines 2-3 — This is the parallel region
Each thread executes code block and subroutine call or function.
No branching (in or out) in a parallel region.

Line 4 — All threads synchronize at end of parallel region (implied barrier).

In example above, user must explicitly create independent work (tasks) in the code block and routine (using thread id and total thread count).

Parallel Region

```
    ...
!$omp parallel

  do i=1,n
    call work(i)
    a(i) = b(i)+c(i)
  end do
!$omp end parallel
```

```
...
#pragma omp parallel
{
  for(i=0;i<n;i++){
    work(i);
    a[i] = b[i]+c[i];
  }
}
```

Ho often are the loop iterations executed?

Note: this code is not 100% correct (see discussion of private variables later)

Parallel Region

```
    ...
!$omp parallel

  do i=1,n
    call work(i)
    a(i) = b(i)+c(i)
  end do
!$omp end parallel
```

```
...
#pragma omp parallel
{
  for(i=0;i<n;i++){
    work(i);
    a[i] = b[i]+c[i];
  }
}
```

Replicated work

In above example the do/for loop iterations are not split among the threads via a do/for work-sharing construct.

Note: this code is not 100% correct (see discussion of private variables later)

Parallel Region with Worksharing Construct

```
    ...
!$omp parallel

!$omp do
  do i=1,n
    call work(i)
    a(i) = b(i)+c(i)
  end do
!$omp end parallel
```

```
...
#pragma omp parallel
{
#pragma omp for
  for(i=0;i<n;i++){
    work(i);
    a[i] = b[i]+c[i];
  }
}
```

Work-sharing

In above example the do/for loop iterations are split among the threads via the do/for worksharing constructs.

Parallel Region & Work-Sharing

Use OpenMP directives to specify Parallel Region, Work-Sharing constructs, and Mutual Exclusion

```
parallel


end parallel
```

Use parallel … end parallel for F90
Use parallel {…} for C

```
parallel do/for
parallel sections
```

Code block	Each Thread Executes
do / for	Work Sharing
sections	Work Sharing
single	One Thread (Work sharing)
master	One Thread
critical	One Thread at a time
atomic	One Thread at a time

A single worksharing construct (e.g. a do/for) may be combined on a parallel directive line.

OpenMP Combined Directives

- Combined directives
  - parallel do/for and parallel sections
  - Same as parallel region containing only do/for or sections worksharing construct

```
!$omp parallel do
  do i = 1, 100
    a(i) = b(i)
  end do
```

trip count required
no exit
cycle ok

```
#pragma omp parallel for
for(i=0;i<100;i++){
  a[i] = b[i];
}
```

trip count required
no break
limited C++ throw.
continue ok

Work Sharing – do/for

Worksharing (WS) constructs: do/for, sections, and single

- **With Work-sharing: Threads execution their "share" of statements in a PARALLEL region.**
- **Do/for Work-sharing may require run-time work distribution and scheduling**

```
1  !$OMP PARALLEL DO
2     do i=1,n
3       a(i)=b(i)+c(i)
       enddo
5  !$OMP END PARALLEL DO
```

```
#pragma omp parallel for
   for(i=0;i<n;i++){
     a[i]=b[i]+c[i];
   }
```

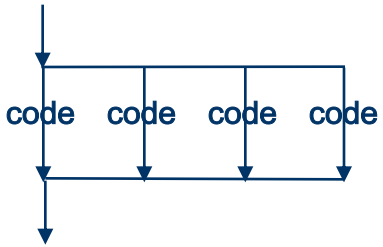
Line 1 Team of threads formed (parallel region).
Line 2-4 Loop iterations are split among threads.
Implied barrier at "enddo" and "}".
Line 5 (Optional) end of parallel loop.

- **Each loop iteration must be independent of other iterations.**

Replicated and Work Share Constructs

- **Replicated:** **Work blocks are executed by all threads.**
- **Work Sharing:** **Work is divided among threads.**

```
PARALLEL
   {code}
END PARALLEL
```



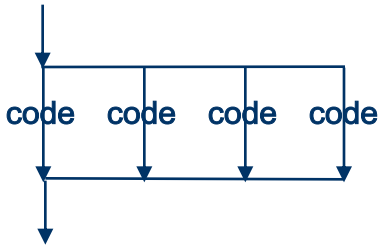
Replicated

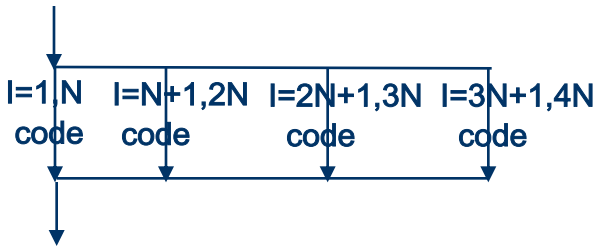
Replicated and Work Share Constructs

- **Replicated:** Work blocks are executed by all threads.
- **Work Sharing:** Work is divided among threads.

```
PARALLEL
  {code}
END PARALLEL
```

```
PARALLEL DO
  do I = 1,N*4
    {code}
  end do
END PARALLEL DO
```





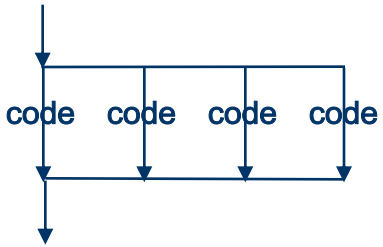
Replicated

Work Sharing

Replicated and Work Share Constructs

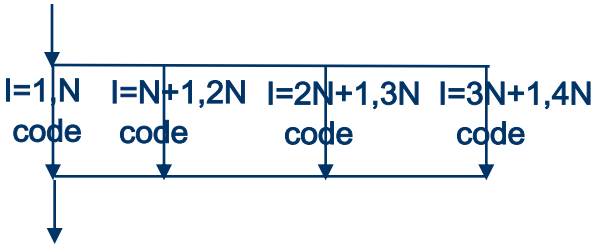
- **Replicated:** Work blocks are executed by all threads.
- **Work Sharing:** Work is divided among threads.

```
PARALLEL
  {code}
END PARALLEL
```



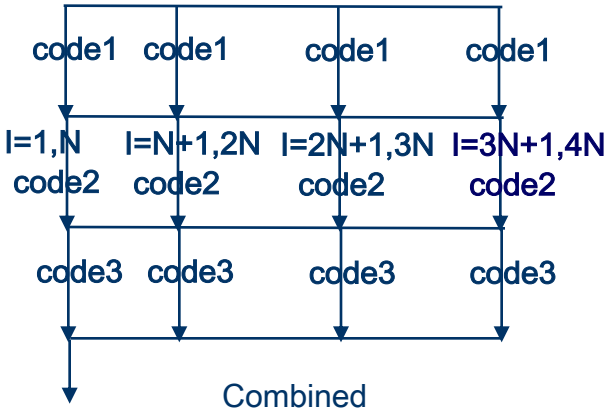
Replicated

```
PARALLEL DO
  do I = 1,N*4
    {code}
  end do
END PARALLEL DO
```



Work Sharing

```
PARALLEL
    {code1}
DO
  do I = 1,N*4
    {code2}
  end do
    {code3}
END PARALLEL
```



Combined

OpenMP Worksharing Scheduling

Clause Syntax: **parallel do/for schedule(*schedule-type* [,*chunk-size*])**

Schedule Type

schedule (static[, chunk])

- Threads receive chunks of iterations in thread order, round-robin. (Divided “equally” if no chunk size.)
- Good if every iteration contains same amount of work
- May help keep parts of an array in a particular processor's cache– good between parallel do/for’s.

schedule (dynamic[, chunk])

- Thread receives chunks as it (the thread) becomes available for more work
- Default chunk size may be 1
- Good for load-balancing

OpenMP Worksharing Scheduling

schedule (guided[, chunk])

- Thread receives chunks as the thread becomes available for work
- Chunk size decreases logarithmically, until it reaches the chunk size specified (default is 1)
- Balances load and reduces number of requests for more work

schedule (runtime)

- Schedule is determined at run-time by the OMP_SCHEDULE environment value.
- Useful for experimentation

OpenMP Worksharing Scheduling

For example, loop with 100 iterations and 4 threads

- schedule(static)

Thread	0	1	2	3
Iteration	1-25	26-50	51-75	76-100

- Schedule(static,10)

Thread	0	1	2	3
Iteration	1-10, 41-50, 81-90	11-20,51-60, 91-100	21-30, 61-70	31-40, 71-80

- schedule(dynamic, 15) *(one possible outcome)*

Thread	0	1	3	2	1	3	2
Iteration	1-15	16-30	31-45	46-60	61-75	76-90	90-100

- schedule(guided, 8) *(one possible outcome)*

Thread	0	1	2	3	3	2	3	1
Iteration	1-25	26-44	45-58	59-69	70-77	78-85	86-93	93-100

Parallel – Worksharing - Schedule

- Combined directives
  - parallel do/for
  - Schedule clause added

```
!$omp parallel do schedule(static,8)
  do i = 1, 100
    a(i) = b(i)
  end do
```

```
#pragma omp parallel for schedule(static,8)
    for(i=0;i<100;i++){
      a[i] = b[i];
    }
```

How will the loop iteration be scheduled?
Assume that the number of threads is 4

OpenMP WorkSharing -- Sections

We will not discuss OpenMP sections in this class

- SECTIONS
  - Blocks of code are split among threads - task parallel style
  - A thread might execute more than one block or no blocks
  - Implied barrier

```
!$OMP SECTIONS

!$OMP SECTION
      CALL TASK1()
!$OMP SECTION
      CALL TASK2()
!$OMP SECTION
      CALL TASK3()

!$OMP END SECTIONS
```

```
#pragma omp sections
{
#pragma omp section
      { TASK1( ); }
#pragma omp section
      { TASK2 ( ); }
#pragma omp section
      { TASK3 ( ); }

}
```

OpenMP Worksharing -- Single

- SINGLE (or MASTER)
  - Block of code is executed only once by a single thread (or the master thread)
  - Implied barrier (only SINGLE, not MASTER)

```
!$OMP single

  glob_count = glob_count + 1
  print *, glob_count

!$OMP end single
```

```
#pragma omp single
{
  glob_count++;
  printf("%d\n", glob_count);

}
```

```
1  !$OMP PARALLEL DO
2     do i=1,n
3      a(i)=b(i)+c(i)
       enddo
5  !$OMP END PARALLEL DO
```

```
#pragma omp parallel for
   for(i=0;i<n;i++){
     a[i]=b[i]+c[i];
   }
```

How many variables i are there?

Logical and 'in memory'

Private variables

Let's go back to our simple parallel loop

Parallel construct

Worksharing (WS) constructs: do/for

```
1 !$OMP PARALLEL DO
2     do i=1,n
3       a(i)=b(i)+c(i)
      enddo
5 !$OMP END PARALLEL DO
```

```
#pragma omp parallel for
   for(i=0;i<n;i++){
     a[i]=b[i]+c[i];
   }
```

Line 1 Team of threads formed (parallel region).
Private copies of **i** are automatically created

Line 2-4 Loop iterations are split among threads.
Implied barrier at "enddo" and "}".

Line 5 (Optional) end of parallel loop.

- Each loop iteration must be independent of other iterations.
- **How many variables are used in the code snippet?**

OpenMP Data Scoping

SHARED - Variable is shared (seen) by all processors.
PRIVATE - Each thread has a private instance of the variable.

Defaults
Fortran: **do** indices are private, all other variables are shared.
C: only OMP workshare **for** indices have private indices.
Both languages: Everything else is shared

```
!$omp parallel do shared(a), &
                  private(t1,t2)
  do i = 1,1000
    t1 = f(i); t2 = g(i)
    a(i) = sqrt(t1*t1 + t2*t2)
  end do
```

```
#pragma parallel for shared(a), \
                    private(t1,t2)
for(i=0; i<1000; i++){
  t1 = f[i];  t2 = g[i];
  a[i] = sqrt(t1*t1 + t2*t2);
}
```

All threads have access to the same storage areas for a, but each thread has its own private copy of the loop index, i, t1, and t2.

OpenMP Clauses -- Scoping

#pragma omp directive-name [*clause* [[,]*clause*]…]
!$omp *directive-name* [*clause* [[,]*clause*]…]

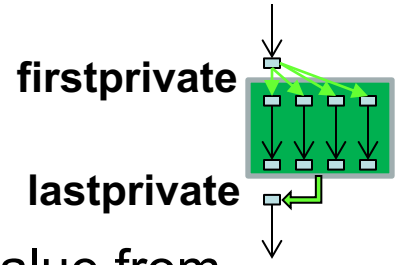
- Data scoping (See section 2.9.3.1-3of OpenMP 3.1 spec.)

  private(*variable list*)
  - Each thread has its own copy of the specified variable
  - Variables are undefined after work sharing region

  shared(*variable list*)
  - Threads share a single copy of the specified variable

  default(*type*)
  - A default of PRIVATE, SHARED or NONE can be specified
  - Note that loop counter(s) of work sharing constructs are always PRIVATE by default; everything else is SHARED by default

OpenMP Data Scoping



- Data scoping (continued)

  firstprivate(*variable list*)

  - Like PRIVATE, but copies are initialized using value from master thread's copy

  lastprivate(*variable list*)

  - Like PRIVATE, but final value is copied out to master thread's copy
  - For/DO: last iteration; SECTIONS: last section

  reduction(*op:variable*)

  - Each thread has its own copy of the specified variable
  - Can appear only in reduction operation
  - All copies are "reduced" back into the original master thread's variable

OpenMP Data Scoping

- Data scoping (continued)
  - do/for and parallel do/for constructs
    - index variable is automatically private
  - non-worksharing loops (nested loops)
    - Fortran: index variable is private (not so in C/C++)
- automatic storage variables
  - private, if with “duration” of scope inside the construct. (e.g. automatic variables in functions)

Variable Scoping, Fortran example

scope

```
program main
integer, parameter :: nmax=100
real*8  :: x(n,n)
integer :: n, j
...
n=nmax; y=0.0
!$omp parallel do
   do j=1,n
      call adder(x,n,j)
   end do
...
end program main
```

lexical extent

```
subroutine adder(a,m,icol)
integer,parameter :: nmax=100
real*8  :: a(m,m)
integer :: i, m, icol
save sum = 0.0    ! C: static
...
do i=1,m
   y(icol)=y(icol)+a(i,icol)
end do

sum=sum+y(icol)
...
end subroutine adder
```

dynamic extent

Worksharing Parallelism

- All about removing dependences between threads:
  - Remove dependences between threads.
  - Remove dependences between iterations (finer parallelism, often see this).
- Accomplished by:
  - Splitting dependences out of loop.
  - Exchange memory for dependences.
  - Exchanging loops
  - … thinking outside of the equations (box)
- Creating more work per thread.

Splitting dependences out of loop

Can these loops be executed in parallel?

```
for(i=1; i<n; i++){
  b[i  ] = a[i  ]*r[i  ]
  a[i-1] = t[i-1]*s[i-1]
}
```

C/C++

```
do i=2,n
  b(i  ) = a(i  )*r(i  )
  a(i-1) = s(i-1)*t(i-1)
enddo
```

F90

Splitting dependences out of loop

Loop fission

```
for(i=1; i<n; i++){
  b[i  ] = a[i  ]*r[i  ]
  a[i-1] = t[i-1]*s[i-1]
}
```

```
for(i=1; i<n; i++)
  b[i  ] = a[i  ]*r[i  ]

for(i=1; i<n; i++)
  a[i-1] = t[i-1]*s[i-1]
```

C/C++

```
do i=2,n
  b(i  ) = a(i  )*r(i  )
  a(i-1) = s(i-1)*t(i-1)
enddo
```

```
do i=2,n
  b(i  ) = a(i  )*r(i  )
enddo
do i=2,n
  a(i-1) = s(i-1)*t(i-1)
enddo
```

F90

Exchange memory for dependences

Exchange Memory for dependence

```
for(i=1; i<n; i++)
   a[i-1] = f(a[i]);
```

```
for(i=1; i<n; i++)
  b[i-1] = f(a[i]);
for(i=0; i<n-1; i++)
  a[i]=b[i];
```

C/C++

```
do i=2,n
   a(i-1) = f(a(i))
enddo
```

```
do i=2,n
   b(i-1) = f(a(i))
enddo
do i=1,n-1
   a(i) = b(i)
enddo
```

F90

The Best Debug Tool

because it is so simple

```
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  printf("tid=%3.3d ...\n",tid);
{
```

C/C++

F90

```
!$omp parallel private(tid)

  tid = omp_get_thread_num()
  write(*,'("tid=",i3.3,"…")') tid
!$omp end parallel
```

```
$ a.out | sort
tid=000 ...
tid=001 ...
...
tid=010 ...
```

Reduction

Let’s go back to our simple parallel loop

Two people calculating: sum(1 to 100)

Reduction

Let's go back to our simple parallel loop

Parallel construct

Worksharing (WS) constructs: do/for

How can we parallelize this correctly?

```
sum = 0
!$OMP PARALLEL
!$OMP DO
do i=1,n
  sum = sum + a(i)
enddo
!$OMP END PARALLEL
```

F90

```
sum = 0;
#pragma OMP PARALLEL
#pragma OMP for
for(i=0; i<n; i++){
  sum = sum + a[i];
}}
```

C/C++

Reduction

F90

Let's go back to our simple parallel loop

Worksharing (WS)

```
integer :: a(n), sum

!$omp parallel

  !$omp do
    do i=1,n
       sum  =  sum      + a(i)
    enddo

!$omp end parallel
```

Reduction

F90

Worksharing (WS) – partial reductions

```
integer :: a(n), sum, psum(nt)

!$omp parallel private(id)

  id = omp_get_thread_num()
  !$omp do
    do i=1,n
      psum(id) = psum(id) + a(i)
    enddo

!$omp end parallel
```

Array for partial sums

Identify Thread

Each Thread's partial sum

Reduction

F90

Worksharing (WS) – partial reduction pre/post processing

```
integer :: a(n), sum=0, psum(nt)
do i=1,nt; psum(i)=0; end do

!$omp parallel private(id)

  id = omp_get_thread_num()
  !$omp do
    do i=1,n
      psum(id) = psum(id) + a(i)
    enddo

!$omp end parallel

do id=1,nt; sum=sum+psum(id); end do
```

Initialize Partial Sum

Reduce Partial Sums

C/C++

Reduction

Worksharing (WS) –

```
int a[n], sum;


#pragma omp parallel
{

  #pragma omp for
    for(i=0; i<n; i++){
        sum     =   sum     + a[i];
    }
}
```

C/C++

Reduction

Worksharing (WS) – partial reduction pre/post processing

```
int a[n], sum, psum[nt];


#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  #pragma omp for
    for(i=0; i<n; i++){
       psum[id] =  psum[id] + a[i];
    }
}
```

Array for partial sums

Identify Thread

Each Thread's partial sum

C/C++

Reduction

Worksharing (WS) – partial reduction pre/post processing

```
int a[n], sum=0, psum[nt];
for(i=0; i<nt; i++) psum[i]=0;

#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  #pragma omp for
    for(i=0; i<n; i++){
       psum[id] =  psum[id] + a[i];
    }
}

for(id=0; i<nt; i++) sum += psum[id];
```

Initialize Partial Sum

Reduce Partial Sums

Reduction in OpenMP

Worksharing (WS) constructs:

Reduction operation

```
sum = 0
!$OMP PARALLEL reduction(+:sum)
!$OMP DO
do i=1,n
  sum = sum + a(i)
enddo
!$OMP END PARALLEL
```

F90

Lets discuss:

How many ‘copies’ of sum exist?
Why do we have to specify the operator?
Why do we have to initialize ‘sum’?

```
sum = 0;
#pragma OMP PARALLEL reduction(+:sum){
#pragma OMP for
for(i=0; i<n; i++){
  sum = sum + a[i];
}}
```

C/C++

Reduction

Let's go back to our simple parallel loop

Parallel construct

Worksharing (WS) constructs: do/for

Reduction operation

```
sum = 0
!$OMP PARALLEL reduction(+:sum)
!$OMP DO
do i=1,n
  sum = sum + a(i)
enddo
!$OMP END PARALLEL
```

F90

```
sum = 0;
#pragma OMP PARALLEL reduction(+:sum){
#pragma OMP for
for(i=0; i<n; i++){
  sum = sum + a[i];
}}
```

C/C++

Lets discuss:

What happens if we initialize sum to 5.?
What happens if we change the reduction operation to 'multiply' (*)?

F90

OpenMP Data Scoping

```
sum = 0
!$omp parallel do reduction(+:sum)
do i = 1, 1000
   sum = sum + a(i)
end do
! Each thread's copy of sum is added
! to original sum at end of loop

!$omp parallel do lastprivate(temp)
do i = 1, 1000
   temp = f(i)
end do
print *, 'f(1000) == ', temp
! temp is equal to f(1000) at end of loop
```

C/C++

OpenMP Data Scoping

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for(i=0;i<N;i++){
    sum = sum + a[i];
}
//Each thread's copy of sum is added
//to original sum at end of loop
printf("sum= %f\n",sum);

#pragma omp parallel for lastprivate(temp)
for(i=0;i<N;i++){
    temp = f[i];
}
printf("f(1000) == %f\n", temp);
//temp is equal to f(1000) at end of loop
```

Debugging

- To debug parallel code -- know semantics of the language
  - The semantics (directives) tells you the restraints on the parallel execution.
  - It is a prescription of behavior of a task relative to other tasks. Know the restraints and what is not constrained.
    - Reread code– think parallel
    - Review: expected behavior vs spec behavior vs implementation behavior – expected behavior does not take into account the unconstrained concurrency.

OpenMP Data Scoping with ‘loop’

Pseudo code

```
count = 0
omp parallel
omp loop
  loop with 2 loop iterations
    count = count + 1
    print ‘count = ‘, count
  end loop
end parallel
```

Serial execution

```
count = 1
count = 2
```

Parallel execution

```
???
```

OpenMP Data Scoping with ‘loop’

Pseudo code

```
count = 0
omp parallel
omp loop
  loop with 2 loop iterations
    count = count + 1
    print ‘count = ‘, count
  end loop
end parallel
```

Serial execution

```
count = 1
count = 2
```

Parallel execution

```
count = 1
count = 1
or
count = 1
count = 2
```

OpenMP Data Scoping with ‘loop’

Pseudo code

```
count = 0
omp parallel 'what to put here?'
omp loop
  loop with 2 loop iterations
    count = count + 1
    print 'count = ', count
  end loop
end parallel
```

`'What to put here?'`

What is the problem if:

- we put nothing here?

OpenMP Data Scoping with ‘loop’

Pseudo code

```
count = 0
omp parallel 'what to put here?'
omp loop
  loop with 2 loop iterations
    count = count + 1
    print 'count = ', count
  end loop
end parallel
```

```
'What to put here?'
```

What is the problem if:

- we put nothing here?
  → race condition

- **`private(count)`**?

OpenMP Data Scoping with ‘loop’

Pseudo code

```
count = 0
omp parallel 'what to put here?'
omp loop
  loop with 2 loop iterations
    count = count + 1
    print 'count = ', count
  end loop
end parallel
```

`'What to put here?'`

What is the problem if:

- we put nothing here?
  → race condition
- `private(count)`?
  → no initialization
- **`firstprivate(count)`?**

OpenMP Data Scoping with ‘loop’

Pseudo code

```
count = 0
omp parallel firstprivate(count)
omp loop
  loop with 2 loop iterations
    count = count + 1
    print ‘count = ‘, count
  end loop
end parallel
```

‘What to put here?’

What is the problem if:

- we put nothing here?
  → race condition
- **`private(count)`**?
  → no initialization
- **`firstprivate(count)`**?
  → are we done?

OpenMP Data Scoping with 'loop'

Pseudo code

```
count = 0
omp parallel firstprivate(count)
omp loop
  loop with 2 loop iterations
    count = count + 1
    print 'count = ', count
  end loop
end parallel
```

'What to put here?'

What is the problem if:

- we put nothing here?
  → race condition
- **`private(count)`**?
  → no initialization
- **`firstprivate(count)`**?
  → are we done?

Serial prints: 1 and 2

Parallel may print: 1 and 1

OpenMP Data Scoping with 'loop'

Pseudo code

```
count = 0
omp parallel
omp loop
  loop with 2 loop iterations
    count = count + 1
    print 'count = ', count
  end loop
end parallel
```

'What to put here?'

What is the problem if:

- we put nothing here?
  → race condition
- **`private(count)`**?
  → no initialization
- **`firstprivate(count)`**?
  → are we done?

No, we need 'something' else

Goals

1. Parallel: print the same regardless of number of threads
2. Print the same for serial and parallel '1 and 2'

TACC

We will cover the 'something else' at a later time

OpenMP “Hello”

```
program hello
use omp_lib

print*, "hello, from master thread“

!$omp  parallel

print*, “id",omp_get_thread_num()

!$omp  end parallel
end program
```

```
#include <omp.h>
#include <stdio.h>
int main(){

printf("hello, from master thread\n");

#pragma omp parallel
  {
   printf("id%d\n",
              omp_get_thread_num());
   }}
```

Compile with intel compiler

PGI compiler	Intel compiler	GNUcompiler
pgf90 -mp hello.f90 pgcc -mp hello.c	ifort -qopenmp hello.f90 icc -qopenmp hello.c	gfortran -fopenmp hello.f90 gcc -fopenmp hello.c

Please run this exercise at home. Be prepared for home work 1

OpenMP “Hello”

- Set Env. Var: export OMP_NUM_THREADS=4
- Execute: ./a.out
- The OpenMP routine omp_get_thread_num() reports unique thread #s between 0 and OMP_NUM_THREADS-1

OpenMP “Hello”

- Output after running on 4 threads :

  hello, main

  thrd=1

  thrd=0

  thrd=3

  thrd=2

- Analysis of OpenMP output :
  - Threads are working completely independently
  - In ‘real’ code threads usually have to cooperate to produce correct results, requiring synchronization

OpenMP Books and Resources

Parallel Programming in MPI and OpenMP by Victor Eijkhout (TACC)
Covers topics in MPI and OpenMP
https://bitbucket.org/VictorEijkhout/parallel-computing-book/downloads

OpenMP books (see picture

Parallel Programming

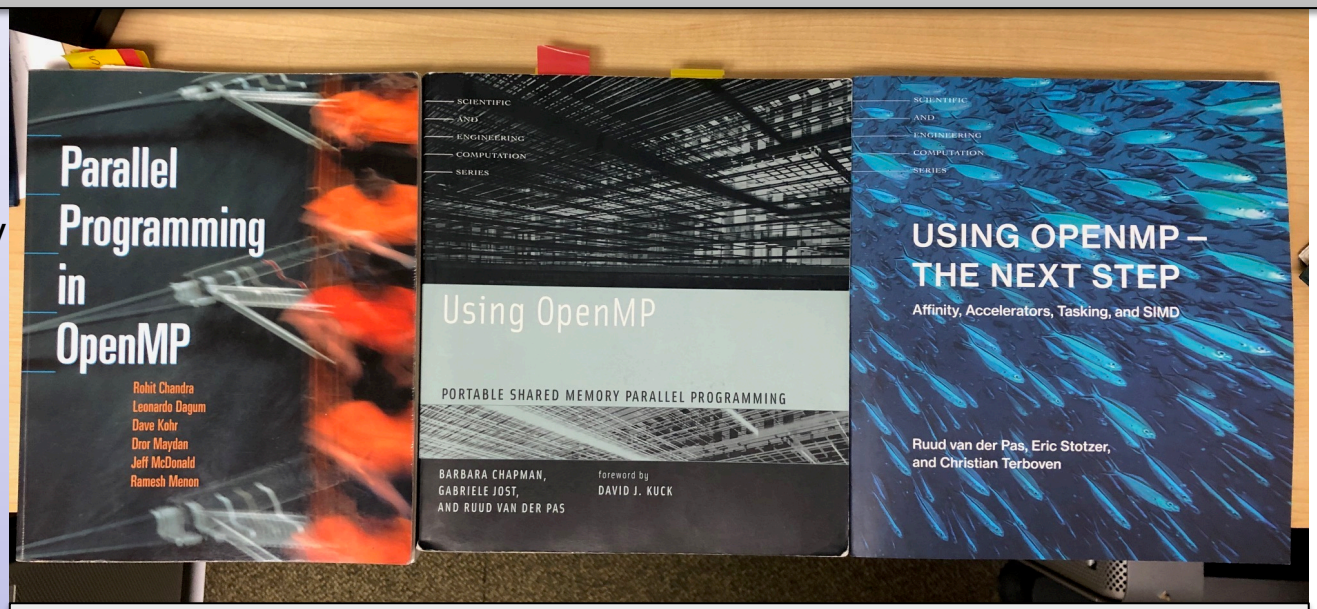
- Little bit older
- Good introduction
- Maybe, if it is cheap to buy used

Using OpenMP

- Covers all topics in class
- Good introduction and advanced topics
- My recommendation for the class

The Next Step

- Part 2 to 'Using OpenMP'
- Advanced topics from recent OpenMP standards



OpenMP resources

www.openmp.org/specifications/
https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5.1-web.pdf

TACC